Some Considerations on the Compile-Time Analysis of Constraint Logic Programs

M.J. García de la Banda
maria@fi.upm.es

M. Hermenegildo
herme@fi.upm.es *or* herme@cs.utexas.edu

Universidad Politécnica de Madrid (UPM)
Facultad de Informática
28660-Boadilla del Monte, Madrid - Spain

Abstract

This paper discusses some issues which arise in the dataflow analysis of constraint logic programming (CLP) languages. The basic technique applied is that of abstract interpretation. First, some types of optimizations possible in a number of CLP systems (including efficient parallelization) are presented and the information that has to be obtained at compile-time in order to be able to implement such optimizations is considered. Two approaches are then proposed and discussed for obtaining this information for a CLP program: one based on an analysis of a CLP metainterpreter using standard Prolog analysis tools, and a second one based on direct analysis of the CLP program. For the second approach an abstract domain which approximates groundness (also referred to as "definiteness") information (i.e. constraint to a single value) and the related abstraction functions are presented.

1 Introduction

The increasing acceptance of constraint logic programming languages has motivated a growing interest in dataflow analysis and implementation techniques for these languages. A constraint logic program is a rule-based program which allows computations over both symbolic and non-symbolic domains by replacing unification with tests for constraint satisfaction. While expressive power can be greatly enhanced in such programs, since the domains used can be richer than the usual Herbrand domain and unification is only a special case of the aforementioned tests for constraint satisfaction, such tests can often be much more expensive than unification and result in low run-time performance.

Two important observations that we would like to point out concerning this issue of CLP performance are as follows: first, that when running general "constraint" programs CLP systems are slower than required for many applications. Second, that current CLP systems are often significantly slower than Prolog systems when running equivalent (i.e. "Prolog") programs. This is not surprising in principle, since of course they are more complex systems with the capability of solving a larger set of problems. However we find this problem to be of particular significance: the most desirable situation is for the CLP system to be the real general purpose problem-solving tool, used to solve all the problems that used to be solved in the Herbrand domain with Prolog plus the new ones allowed by the new constraint solving capabilities. However, if the performance in the Herbrand domain (i.e.

when "running what Prolog can run") is too low compared with that of current Prolog systems, and given the very large set of applications for which Prolog has proved itself quite adequate, users will tend to have both a state-of-the-art Prolog system and a CLP systems, a less than optimal situation, apart from discouraging writing applications that work on a combination of domains.

Our objective in the compile-time analysis of CLP programs will then be twofold: first, to correctly identify CLP programs that are actually "Prolog" programs (or those parts of CLP programs that stay within the Herbrand domain and operate "as Prolog") and make the CLP implementation competitive for these programs (by using standard Prolog implementation techniques [33]). Second, and in line with the aims of conventional analysis tools, to optimize the non-Herbrand parts by gathering dynamic information about the program statically in order to use more efficient, specialized versions of the constraint solving algorithms. For this, we propose using, among other techniques, abstract interpretation.

Much work has been done using the abstract interpretation technique in the context of logic programs. There, it has been applied to the achievement of several types of high level optimizations: mode inference analysis [11, 26], efficient backtracking [3], garbage collection [23], aliasing analysis [18, 28, 27], type inferencing [24, 2], etc. and it has shown great usefulness and practicality [32, 31, 24, 34]. Using the results from abstract interpretation to produce specialized versions of predicates for different run-time instantiation situations [12, 13] and this combined with invariance detection [14, 35] have also been proposed. It appears possible that some of those types of optimizations are also applicable to constraint logic programming languages. In addition, the characteristics of those languages suggest new needs and types of optimizations. Exploring these possibilities is the subject of this paper.

In the context of abstract interpretation based compile-time analysis of CLP programs, a few general frameworks have recently been defined [25, 4]. However, it is yet not clear what particular compile-time information is actually required to detect situations in which optimization opportunities arise. Also, it is unclear how conventional tools have to be modified (if possible at all) to safely and accurately approximate such information.

In this paper we provide a first step in this direction. Several types of optimizations possible in some constraint logic programming systems and the information that has to be obtained at compile-time in order to implement such optimizations are discussed. Two approaches are then proposed and considered for obtaining this information: one based on an analysis of a constraint logic program metainterpreter using standard Prolog analysis tools, and a second one based on direct analysis of the constraint logic program. For the second approach an abstract domain which approximates groundness (i.e. constraint to a unique value, also referred to as "definiteness" in this context [25, 4]) information and its abstract functions are presented.

The rest of the paper procceeds as follows: section 2 recalls the basic concepts of Constraint Logic Programming and Abstract Interpretation, and presents some of the optimization opportunities that we have identified and have motivated our work. Section 3 presents the two approaches proposed to the analysis of constraint logic programs and discusses them. Section 4 presents an abstract domain for groundness and its abstract functions, and finally, section 5 presents our conclusions.

2 Preliminaries

Although the reader is assumed to be familiar with the basic notions in Logic Programming, Constraint Logic Programming, and Dataflow Analysis, in order to be self contained, the following subsections briefly summarize those concepts.

2.1 The Constraint Logic Programming Paradigm

In this section we present some basic concepts of Constraint Logic Programming. We follow mainly [19] and [20]. The Constraint Logic Programming Paradigm is an extension of the Logic Programming Paradigm in which unification is replaced by the concept of constraint solving performed over an interpreted structure not restricted to the Herbrand Universe. Programs under this scheme are provided with an algebraic semantics. Let us first consider this algebraic point of view.

Let $\mathcal{B}$ be a set of sorts, Σ a $\mathcal{B}$-sorted set of function symbols, $\mathcal{X}$ a $\mathcal{B}$-sorted set of variables, $\Pi = \Pi_C \cup \Pi_P$ a $\mathcal{B}$-sorted set of predicate symbols, where Π_C are the constraint predicates including the symbol "=" and $\Pi_C \cap \Pi_P = \emptyset$

Let (Σ)-terms and $(\Sigma \cup \mathcal{X})$-terms be the ground and possibly non ground terms respectively, $(\Pi_C, \Sigma \cup \mathcal{X})$-atoms be the atomic constraints, and $(\Pi_P, \Sigma \cup \mathcal{X})$-atoms be simply atoms. A constraint is a (possibly empty) set of atomic constraints that will be interpreted as the conjunction of its elements.

A Constraint Logic Program is a finite set of constraint rules of the form:

$H \leftarrow \pi \Box .$
or
$H \leftarrow \pi \Box B_1, ..., B_n.$

where π is a finite constraint and H, $B_1, .., B_n$ are atoms. A goal is a constraint rule with no head and with a non empty body.

The structure defined on Π and Σ, will be denoted by $\Re(\Sigma, \Pi)$ and consists of an interpretation domain $\mathcal{D}_f$ for each f belonging to $\mathcal{B}$, and an interpretation over$\{\mathcal{D}_f\}$ for the function symbols in Σ and the constraint predicates in Π_C.

An $\Re(\Sigma, \Pi)$-valuation v of a $\Re(\Sigma, \Pi)$-expression E, is a mapping from each distinct variable in E, into $\mathcal{D}_f$ respecting their sorts. If v makes true in $\Re(\Sigma, \Pi)$ all constraint atoms of a constraint c, it is called an $\Re(\Sigma, \Pi)$-solution of c. A constraint is $\Re(\Sigma, \Pi)$-solvable iff there exists an $\Re(\Sigma, \Pi)$-solution for it.

From the logical interpretation point of view, a program would be a first order theory dealing with the usual first order formulas. In order to do that, the first order theory has to correspond to a structure $\Re(\Sigma, \Pi)$ (in the following $\Re$) which has to be solution compact. A first order theory $\Im$ corresponds to a structure $\Re$ if (1) for each constraint π which has solution in $\Re$, $\Im \models \exists \pi$ and (2) $\Re \models \Im$. A structure $\Re$ is said to be solution compact if (1) any element of the structure is the solution of a finite or infinite set of constraints and (2) an infinite set of constraints is unsolvable iff a finite subset of it exists which is unsolvable.

Let us now consider the operational semantics of constraint logic programming languages. Let P be a constraint logic program, A $(P, \Re)$-derivation step of a goal G: $\leftarrow \quad c \Box Q$ where $Q = ..., A_{i-1}, A_i, A_{1+1}, ...$ (a nonempty goal is needed) returns a goal of the form $G' :\leftarrow \quad c' \Box Q'$ being $Q' = .., A_{i-1}, B, A_{1+1}, ...$ if there exists a variant of a clause in P, $C \quad = \quad H \leftarrow b \Box B$, such that:

- A_i and H have the same predicate symbol,
- C has no variables in common with G,
- $c' = c \wedge b \wedge (Q = B)$ is $\Re$-solvable

Then G' is the *resolvent* of the derivation step. The condition can be relaxed asking c' to be an $\Re$-solvable restriction of $c \wedge b \wedge (Q = B)$ called a generalized derivation step. In particular any derivation step is a generalized one. The $(P, \Re)$-derivation of a goal G is a finite or infinite sequence of goals returned by $(P, \Re)$-derivation steps, starting from G. It will be successful when an empty sequence of atoms is reached, i.e. the last goal contains only constraints, which are the *answer constraints* to the original query. Finitely failed sequences are those in which no goal can be expanded.

Once the Constraint Logic Programming Paradigm has been considered, the next step is to introduce the concept of constraint system, and some instances of them which have been widely used by current languages.

2.2 Actual Languages and their Constraint Systems

Let $\Re$, Σ, Π and $\mathcal{X}$ be defined as above. A *constraint system* is a quadruple $S = (\Re, I, \mathcal{X}, \Phi)$, where I is an $\Re$-interpretation and Φ a non empty subset of $(\Pi_c, \Sigma \cup \mathcal{X})$-atoms closed under conjunction. We will say that a constraint c is consistent iff there exist an $\Re$-valuation θ such that $I(c)\theta$ is equivalent to true. An *entailment relation* $\vdash$ between constraints is defined by: $c_1 \vdash c_2$ iff c_2 is consistent whenever c_1 is consistent.

The Constraint Logic Programming Framework defines a class of languages. Each of them is an instance of this framework obtained by the specification of its constraint systems. It allows having a numerical and/or logical character to be tackled in a declarative way, i.e. state properties directly in the domain of discourse instead of having them coded, with non-deterministic computation.

There are several existing constraint logic programming languages such as CLP($\Re$)[21], CHIP [22], PrologIII [5], BNRProlog [29], etc. that aside from the underlying Herbrand structure (i.e. the Herbrand Universe with constraints involving only the equality predicate), which they share, differ in their remaining constraint systems.

The domain of computation of CLP($\Re$) is real numbers, with linear equations handled by Gaussian elimination and inequalities solved by an adaptation of the simplex algorithm. Non-linear equations are simply delayed until they become linear.

PrologIII has three different domains of computation namely: rational terms with equations, disequations and inequalities solved by a symbolic simplex-like algorithm when linear equations are considered, boolean terms with equalities solved by a saturation method combined with SL-resolution, and infinite trees with a construction function and $=$ and $\neq$ relations.

CHIP has also three different domains of computation. Rational terms are handled in a similar way than PrologIII but with a different symbolic simplex-like algorithm at the solver-level. Boolean terms with equalities are solved by a boolean unification algorithm. Finally it allows computations over finite domains in which constraints are defined by subsets of the Cartesian product of each finite variable domain allowing not only arithmetical constraints but also symbolic ones and even user defined constraints. Since most discrete combinational problems are NP-complete, for which no general and efficient solving method exists, no complete constraint-solver is used. Consistency techniques are used to prune search space, being the user's responsibility to decide the strategy to apply.

On the other hand BNR-Prolog has focused on the use of local propagation techniques on constraint networks of closed intervals on the real line. It allows to solve simultaneous non-linear equations by systematically narrowing constrained intervals. Since the algebraic properties of expressions in interval arithmetics is preserved, combining symbolic techniques with numerical techniques is possible.

2.3 Optimization Opportunities

As we have seen, constraint logic programming languages are based on different constraint solvers (which are mostly defined as "black-boxes," except perhaps for the finite domains in CHIP which are directly under user control) each of which operates on its own constraint system. It is clear that as the number of constraints handled increases, tests for constraint satisfaction become significantly expensive, decreasing the performance of the execution. Therefore, it is essential to optimize them by reducing the amount of work involved. Obviously, useful ways of optimizing the constraint satisfaction process include avoiding unnecessary constraint solver calls, avoiding constraint satisfaction tests (i.e. adding equations known to be consistent without tests), improving constraint solver computations, etc. In particular, and as mentioned in the introduction, a potentially quite useful type of optimization is to identify constraints (or whole programs) which operate exclusively on the Herbrand domain since the quite successful techniques used in Prolog compilation [33] (which are themselves for the most part the application of specialized cases of the unification algorithm to special cases identified at compile-time) can then be applied.

As mentioned before the optimizations mentioned so far in general try to reduce the amount of work to be done in solving the constraints. An independent and complementary way of improving performance is parallelizing the program, i.e. identifying parts of the work that *needs* to be done which are in some sense independent and can thus be executed in parallel.

In this section we will present a number of possibilities which lead to either reduction of the work to be done or to the identification of opportunities for parallelism in CLP systems. We are not going to make an exhaustive presentation of all different optimizations which can be achieved, but rather mention a few that have motivated our work, together with the compile time information which we have identified as needed to apply them.

One difficulty in discussing such optimizations is the diversity of constraint systems possible. In order to take into account this diversity, we will divide the types of optimization into three broad categories: general, i.e. useful for all languages, specialized for a particular constraint system, or

specialized for a particular constraint solver. Because of lack of space the types of optimizations specifically related with the Herbrand Universe will not be discussed. Several references were given in the introduction to the work done for this domain in the context of traditional logic programming.

With respect to general optimizations, there are at least two cases in which constraint solver calls can be avoided. First, any constraint in which all variables are ground, i.e. constrained to a unique value, can be turned into a simple check. Consider the well known *mortgage* program:

$$
\begin{aligned}
&mortgage(X_p, X_t, X_i, X_r, X_b) :- \\
&\qquad X_t = 1, \\
&\qquad X_b + X_r = X_p * (1 + X_i/1200). \\
&mortgage(X_p, X_t, X_i, X_r, X_b) :- \\
&\qquad X_t > 1, \\
&\qquad X'_p = X_p * (1 + X_i/1200), \\
&\qquad X'_t = X_t - 1 \\
&\qquad mortgage(X'_p, X'_t, X_i, X_r, X_b).
\end{aligned}
$$

If we know that, in the query, the second argument X_t is constrained to a unique value and we are able to infer, by groundness propagation, that in the recursive calls this argument remains ground, each $X_t > 1$ and $X_t = 1$ can be transformed into ground checks (equivalent to Prolog's), thus avoiding actual constraint solver calls.

Second, any constraint in which one of its sides is a free variable (i.e. a variable which has not been involved in any constraint relation up to renaming with other free variables), can be transformed into an assignment if its predicate symbol is "=" or added to the system as a consistent constraint in any case. Consider the variables X'_p and X'_t in the mortgage example. Both are new variables, and therefore they have never been constrained. Thus, the equations in every recursive call can be transformed into an assignment.

Another issue is finding advantageous constraint orderings which improve the efficiency of the constraint solver. For example, by placing most "constraining" constraints first, large failed computations can be avoided (the "first fail principle" illustrates this point). Informally, we can say that a constraint is more "constraining" than another if the number of free variables is less than in the other or if, being equal, one has more variables constrained to a unique value. In order to do that, we do not need information about concrete variable values, just about groundness and freeness. Consider the following set of constraints in a PrologIII program:

$$...., L :: N, U :: N1, L = U. < X > .U, ...$$

in which "." represents list concatenation and '::" is a list length predicate symbol. If we know that at this point of the execution the list U will be ground while the list L will not, it would be better to change the order of the constraints in such a way that the length constraint over U would be computed first. Then if it fails, the length computation over L would be avoided.

When related to a particular constraint system, the accuracy of the optimizations depends on the accuracy of the information, i.e. on the knowledge available about the constraint system. A useful optimization that does not require highly specialized information can be achieved for arithmetical linear constraints. Every time a linear constraint has a variable that has never been constrained, the constraint can be added to the consistent system without testing for constraint satisfaction, i.e. the system remains consistent after adding the constraint. Obviously, the coefficient of the free not constrained variable has to be different form zero. Consider again the *mortgage* example. If the variable X_r or X_b is a free variable in the query, and either X_p or X_i are ground, the last equation is always consistent. Consider also the following clause used in an *add lists computation* in PrologIII:

$$
\begin{aligned}
&add\_lists(<>, <>, <>). \\
&add\_lists(< X > .Xs, < Y > .Ys, < Z > .Zs) :- X + Z = Y, add\_lists(Xs, Ys, Zs).
\end{aligned}
$$

If we know that one of the lists is a free variable and we can infer that then all their elements will be free variables, all equations can be added to the system without constraint satisfaction tests. Note that this optimization is not general as, for example, it does not work for list constraints: consider the following list constraint in PrologIII:

$...., < 1,2,3 >=< 4,5,4 > .Y, ...$

it is clear that, even if Y is a free variable, the constraint will fail.

Another quite useful optimization which is dependent of the constraint system is to eliminate constraints that can be known at compile time to always succeed or fail without modifying the store, i.e. that are "abstractly executable" as defined in [14]. These can also be referred to as "redundant constraints" and are often found for example in programs which do argument type checking. For instance, consider the constraint $X = Y + Z$, if we know that X is bound to a Herbrand Term at this point of the program, we can say that this goal will always fail, and therefore replace it with "fail", eliminate all subsequent goals in the clause, and, for a pure clause, even eliminate the clause in which the goal appears as "dead code". Consider also the constraint $var(X)$, if we know that X is free at this point of the program, we can say that it will always succeed, and therefore eliminate the constraint.

As we have seen, the above simplification can be achieved with quite simple information about X. However, if we want to extend the number of constraints that can be abstractly executable, in some cases much more specialized information is required. Consider the constraint $Y > 1$, in order to know at compile time that this check is going to fail or succeed, we would have to know the concrete value or interval of concrete values that this variable is going to be constrained to at run time, for a given query. Note that this means having information about a possibly infinite domain of values, as for example in the case of real numbers.

In general, with the right type of information multiple optimizations are possible such as avoiding unnecessary constraints, eliminating redundant tests in loops , avoiding choice points, eliminating dead code, detecting errors, etc.

It is important to note that, in order to obtain at compile time the information needed for the optimizations we have pointed out, global analysis rather than local analysis is needed. Motivation for this is that the information has to be propagated throughout all the program, since complete information cannot be inferred just inside the scope of a single clause.

As mentioned before, another way of improving the execution speed of constraint logic programs is by parallelization. Due to its importance and complexity, parallelization of constraint logic programs as a constraint system dependent optimization will be explained in the next subsection.

2.4 Parallelization: Constraint Independence

There has been significant interest in parallel execution models for logic programs. Different computational models have been proposed to exploit either Or-parallelism, And-parallelism, or a combination of both [6]. Motivation for this, is that these models often offer performance improvements while at the same time preserving the conventional "don't know" semantics of logic programs and the computational complexity expected in the execution by the programmer [16].

In the Or-parallelism model the clauses of a predicate are tried in parallel, while in the And-parallelism the goals which are executed in parallel are those of a clause body. There are two forms of And-parallelism: Independent And-parallelism and Stream And-parallelism. The former allows parallelism among goals if they are independent, i.e. do not restrict each other search space. The latter allows non independent goals to be executed in parallel with the value of the dependent variable being communicated incrementally among goals.

As far as we know, only the Or-parallelism model has been studied for the constraint logic programming framework, resulting in an implementation for the CHIP language [15]. The aim of this section is to provide a first step in the study of the And-parallelism, and in particular the Independent And-parallelism model, for constraint logic programming languages. The main problem in the Independent And-parallelism model is to determine which goals are independent and therefore eligible for parallel execution and which goals have to wait for each others during the execution. Although it can be done at run-time, it implies significant overhead. As is explained in [16], most of the work can be performed at compile-time by using dataflow analysis, and in particular abstract interpretation.

In order to do that, we will give a rather informal description of the idea in the model considered. This is as follows: partition the given resolvent so as to obtain some new partial resolvents and a remaining part, execute the partial resolvents, and then embed the information gathered from such execution into the remaining part.

Let us explain more precisely the two frameworks. Assume G: $\leftarrow \ c \ \Box Q_1, ..., Q_n$ is the current resolvent. The sequential proof procedure with left-to-right selection rule would

- select the atom Q_1 and compute the derivation step $G_1 :\leftarrow \ c_1 \ \Box \ B_1, Q_2, ..., Q_n$ where $c_1 \equiv c \wedge_1 \wedge Q_1 \ = \ B_1$ is $\Re$-solvable,
- select the atom Q_2 and compute the derivation step $G_2 :\leftarrow \ c_2 \ \Box \ B_1, B_2, ..., Q_n$ where $c_2 \equiv c_1 \wedge b_2 \wedge Q_2 \ = \ B_2$ is $\Re$-solvable,

and so on until the empty sequence of atoms will be reached.

If, on the contrary, we want to select some of the atoms in parallel, say Q_i and Q_j (the extension to more than two goals is straightforward), one possible execution scheme for G could be the following:

- partition G into the partial resolvents:
  - $R_1 :\leftarrow \ c \ \Box Q_i$
  - $R_2 :\leftarrow \ c \ \Box Q_j$
  - $R_3 :\leftarrow \ \pi \ \Box Q_1, ...Q_{i-1}, Q_{i+1}, ..Q_{j-1}, Q_{j-1}, ...Q_n$
- execute R_1 and R_2 in parallel obtaining the answer constraints $\pi_1 \equiv \ c \wedge b_i \wedge (Q_i = B_i)$ and $\pi_2 \equiv \ c \wedge b_j \wedge (Q_j = B_j)$.
- test for constraint satisfaction of the conjunction $\pi = \pi_1 \wedge \pi_2$
- execute R_3.

Our aim is, within the execution model presented, to run in parallel as many goals as possible while maintaining correctness and efficiency. This means that given a goal G, of which we know the resulting answer constraint of its sequential execution and the time complexity to obtain it, we would like to execute some of the goals in G in parallel obtaining an $\Re$-solvable restriction of the answer constraint obtained before, possibly in a shorter time.

However, using the standard conjunction of constraints in the third step above can lead to unnecessary computations, as shown by the following example in CLP($\Re$):

$\mathrm{p}(X, Y, Z) : -X > Y, X > Z, q(Y), q(X), q(Z).$

q(1).
q(2).

Let $c = X > Y \wedge X > Z$ be the consistent constraint before q(Y) is executed. In this case, the sequential execution of G returns the derivation step $G_1 :\leftarrow \ c_1 \ \Box \ q(X), q(Z)$ with $c_1 = X > Y \wedge X > Z \wedge Y = 1$. During the computation of the second atom, the constraint satisfaction test for $c_2 = X > Y \wedge X > Z \wedge Y = 1 \wedge X = 1$ will fail. Therefore, after backtracking the next derivation step would be $G_2 :\leftarrow \ c_2 \ \Box \ q(X)$ with $c_2 = X > Y \wedge X > Z \wedge Y = 1 \wedge X = 2$, and finally the empty goal will be reached returning the answer constraint $c_3 = X > Y \wedge X > Z \wedge Y = 1 \wedge X = 2 \wedge Z = 1$.

On the other hand, in the parallel execution model we could partition G into the partial resolvents $R_1 :\leftarrow \ c \ \Box q(Y)$, $R_2 :\leftarrow \ c \ \Box q(X)$ and $R_3 :\leftarrow \ c \ \Box q(Z)$, and the remaining part $R_4 :\leftarrow \ \pi \ \Box$. If executed in parallel the partial resolvents R_1, R_2 and R_3 will return the answer constraints $\pi_1 = X > Y \wedge X > Z \wedge Y = 1$, $\pi_2 = X > Y \wedge X > Z \wedge X = 1$ and $\pi_3 = X > Y \wedge X > Z \wedge Z = 1$ respectively. When conjunction of them is considered in order to obtain π, the resulting constraint is not consistent and more backtracking steps are then needed to achieve the same answer constraint as that of the sequential computation.

On the other hand, assume that in the above example the atom $p(X)$ did not appear in the clause. We could partition the goal into $R_1 :\leftarrow \ c \ \Box q(Y)$, $R_2 :\leftarrow \ c \ \Box q(Z)$ and $R_3 :\leftarrow \ \pi \ \Box$. Then the resulting constraint from the parallel execution of R_1 and R_2 would be $\pi = X > Y \wedge X > Z \wedge Y = 1 \wedge Z = 1$ which is consistent and no extra backtracking would be required.

It turns out that some conditions have to be satisfied when executing goals in parallel, in order to ensure the independence of those goals. As we have said, a set of goals are said to be independent if they do not restrict each other's search space in the sequential execution. When only unification is considered within the Strict-Independent-And-Parallelism model, it can be described as those sets of goals which do not share variables at execution time [16]. If the Non-Strict-Independent-And-Parallelism model is considered, it can be described as those goals which are either strict independent or with only one goal instanciating shared variables [17]. However, when full constraint solving is considered, this notion of independence among goals needs to be generalized.

Let $S = (\Re, I, \mathcal{X}, \Phi)$ be a constraint system and L a constraint logic programming language defined over it. Let P be a program in this language and $G :\leftarrow c \Box g_1, .., g_n$ be a goal with $c = c_1 \wedge, ..., \wedge c_n$ being the current consistent constraint in S. Let $var(g_i)$ and $var(c_i)$ be the set of variables involved in the ith-goal and in the ith-constraint respectively, which are not constrained to a unique value.

Definition 1 (constraint independence for variables) *Two variables X,Y, $X \neq Y$ are constraint independent in c iff $\neg\exists a, b \in \Phi$, $\{X\} = var(a)$, $\{Y\} = var(b)$ such that $c \wedge a$ is consistent, $c \wedge b$ is consistent and $c \wedge a \wedge b$ is inconsistent.* $\Box$

Definition 2 (strict constraint independence for goals) *The goals $g_1, .., g_n$ are constraint strictly independent for c iff $\forall X \in var(g_i)$, $\forall Y \in var(g_j)$, $\forall i, j$, $1 \leq i, j \leq n$, $i \neq j$, X and Y are constraint independent.* $\Box$

Let us apply those concepts to the example given above. If we look at the definition, it is clear that the three goals could not be run in parallel: neither the constraints $Y = 1, X = 1$ nor $Z = 1, X = 1$ satisfy the definition. However, there was no entailed constraint relating Y and Z (recall that the predicate symbol ">" establishes an antisymmetrical relation and therefore Y and Z are not related by transitivity), and therefore the conjunction of any consistent constraint over Y and any consistent constraint over Z would also be consistent. As a result, if $q(X)$ did not appear in the clause, $q(Y)$ and $q(Z)$ could be run in parallel.

As we have said, the condition can be extended to "non-strict constraint independence" in the spirit of [17]. Informally, a set of goals at some point of the execution with a consistent constraint c can be *non-strict constraint independently* run in parallel if they are strictly independent or if only one goal adds new constraints involving constraint dependent variables on c. More formally:

Definition 3 (non-strict constraint independence for goals) *Let S L, P, and G, as before. The given goals are non-strict constraint independent for c if:*

1. *they are strict constraint independent or*
2. *if $\forall X \in var(g_i)$, $\forall Y \in var(g_j)$, $\forall i, j$, $1 \leq i, j \leq n$, $i \neq j$, such that X and Y are constraint dependent, $\exists!$ g_l, $l : 1, ..., n$ which adds new constraints over X and Y.* $\Box$

This condition is more relaxed than the previous condition although still not necessary. Note that in order to infer these conditions at compile time, information about the constraint relations among variables is needed. Motivation for this is that, as we have seen in the above example, if no entailed constraint exists over two variables in the system of constraints then these variables are constraint independent. Formally:

Theorem 2.1 *Two variables X,Y, $X \neq Y$ are constraint independent in c if $\neg\exists$ $k \in \Phi$ $X, Y \in var(k)$, such that $c \vdash k$.*

Proof 1 *Let us reason by contradiction. Suppose X, Y are dependent variables. Then $\exists a, b \in \Phi$, $\{X\} = var(a)$, $\{Y\} = var(b)$ such that $c \wedge a$ is consistent, $c \wedge b$ is consistent, and $c \wedge a \wedge b$ is inconsistent. Since $var(a) \cap var(b) = \emptyset$, $a \wedge b$ is always consistent. Therefore there must be a constraint k, $c \vdash k$ such that $k \wedge a \wedge b$ is inconsistent. If $X \notin var(k)$ then $k \wedge a$ is consistent and therefore $k \wedge b$ must be inconsistent, which is impossible. If $Y \notin var(k)$ then $k \wedge b$ is consistent and $k \wedge a$ must be inconsistent what is also impossible. Therefore we have found a constraint k such that $X, Y \in var(k)$, and $c \vdash k$, i.e. a contradiction with the hypothesis.*

Consider for example the *add lists* program presented in section 2.3. In order to preserve correctness we have to infer not only that the variables X,Y and Z are related by arithmetical constraints, but also that those bounded to lists are also related although they do not share any element. Once again, in order to obtain this information, global dataflow analysis is needed. Recall that, local analysis cannot detect constraint relations established outside of the scope of the considered predicate.

However, this sufficient condition lose some information, i.e. it is too conservative. Consider the following constraint c:

$$c \equiv\ Y + Z > X\ \wedge\ X > U + V\ \wedge X = 3$$

Although $Y + Z > U + V$ is entailed by c, once X has been constrained to a unique value, later added constraints over Y or Z can affect neither U nor V. Therefore, in order to obtain more accurate information, we have to take into account when the variables which allow the entailment become ground.

2.5 Program Dataflow Analysis using Abstract Interpretation

In the last two sections we have presented the optimization opportunities which have motivated our work. As we have seen, those optimizations can be achieved by providing the required information to the compiler. How to capture this information will be the subject of the rest of the paper.

Detecting program errors and optimizing programs are tasks of a wide range of programming tools. Dataflow analysis provides essential information to them by deciding whether some *invariant* holds at some *program point* [30]. However, accurate dataflow analyses are difficult to design and prove correct.

The "abstract interpretation" framework first proposed by Cousot and Cousot in [8] provides the basis for a semantics-based approach. Dataflow analysis is described by constructing an *abstract* semantics which approximates a *concrete* collecting semantics (which typically expresses the gathering of the information which is required for the analysis) and for which the meaning of a program is finitely computable. Informally, it is viewed as an *"approximate computation"* performed over an *abstract domain* (descriptions of data) rather than over the data themselves. The abstract semantics is shown to approximate the concrete semantics, and under appropriate conditions to provide dataflow information, thus providing the basis for discussing dataflow analysis correctness.

Data and data-descriptions are related via a pair of functions referred to as the *abstraction* (α) and *concretization* (γ) functions. In addition, each primitive operation u of the language (for example, unification in traditional logic programming) is abstracted to an operation u' over the abstract domain. Soundness of the analysis requires that each concrete operation u be related to its corresponding abstract operation u' as follows: for every x in the concrete computational domain, $u(x)$ is "contained" in $\gamma(u'(\alpha(x)))$.

To ensure that the abstract meaning of a program will be finitely computable, the abstract domain and the abstract functions are chosen in such a way that the Kleene sequences for the functions are finite. This need has been usually achieved by choosing abstract domains which were complete lattices or cpo which is ascending chain finite. However, the domain could be an infinite lattice if there are functions that enforce or accelerate the convergence of the fixpoint function. In this context the widening/narrowing approach has been recently compared to the Galois approach in [9, 7]. It seems that this approach can significantly improve the precision of the analyses, being very useful whenever the required information about the concrete domain is not a finite abstraction as groundness or freeness but a concrete value or an interval out of a possibly infinite set of concrete values.

3 Constraint Logic Programming Analysis: Approaches

The analysis of constraint logic programming languages is based on constraint satisfaction. However, as we have seen, most constraint solvers are defined as "black boxes", i.e. are implemented in other languages (like C), and are neither visible to the user nor to the analyzer. Two alternatives to their analyses are herein suggested namely: to make the constraint solver visible to the analyzer as part of the program, or keep it as a "black box" being the needed information assumed by the analyzer. Both approaches are described in the following subsections.

3.1 Using Standard Prolog Tools

The first approach consists in implementing constraint logic programs in standard Prolog by developing constraint solvers in Prolog and adding two new parameters that will be passed over the whole program: an input and an output list of constraints. Consider the following constraint logic program over the reals with only equations:

$$p(X,Y) : -X + Y = 6, X - Y = 2.$$

Now, consider the following translation of this constraint logic program implemented in standard Prolog:

$$p\_prolog(X,Y,IC,OC) : -add(X,Y,6,IC,OC1),\ subst(X,Y,2,OC1,OC).$$

where IC = "input constraints", and OC = "output constraints", and

$$\begin{array}{ll} add(X,Y,Z,C,C) & :- number(X),\ number(Y), Z\ is\ X+Y. \\ add(X,Y,Z,C,C) & :- number(X),\ number(Z), Y\ is\ Z-X. \\ add(X,Y,Z,C,C) & :- number(Y),\ number(Z), X\ is\ Z-Y. \\ add(X,Y,Z,CI,CO) & :- (var(Y),\ var(Z);\ var(X),\ var(Y);\ var(X),\ var(Z)), \\ & \quad test([Z = X+Y|CI], CO). \end{array}$$

being $test/2$ the linear equation solver implemented in Prolog. The code for the *subst* operator is similar.

The interesting point is that at this point, we are able to analyze the "constraint" logic program with standard prolog tools. This has a double advantage: first, we can concievably infer information about the original CLP program from this analysis. Second, we can infer information about the Prolog program itself which might be useful for simplifying it. If the simplification is extensive enough (including generation of specialized versions for different predicate call patterns and partial evaluation) the resulting program may even be a quite efficient implementation of the original CLP program. This is specially the case if the CLP program was, for the query mode analyzed, actually a "Prolog" program and the analysis is powerful enough to detect it, since the result of the transformations would be the CLP program itself efficiently running on Prolog as a Prolog program.

For example, if we are able to infer that the first argument X of the query is going to be constrained to a unique value, then only the second clause of $add/5$ (and $subt/5$) will be applicable. Then we can specialize for this particular case obtaining the following:

$$p\_prolog(X,Y,C,C) : -add(X,Y,6,C,C),\ subst(X,Y,2,C,C).$$

$$\begin{array}{ll} add(X,Y,Z,C,C) & :- Y\ is\ Z-X. \\ subst(X,Y,Z,C,C) & :- Y\ is\ X-Z. \end{array}$$

which can be transformed into:

$$p\_prolog(X,Y) \quad :- Y\ is\ 6-X,\ Y\ is\ X-2.$$

which bypasses the constraint solving. Therefore, avoiding constraint solver calls could be achieved by optimizing the constraint solver interface for some call patterns. Furthermore, all the improvements that have been achieved by abstract interpretation in the context of logic programs can be applied to the Prolog program, including the constraint solver.

The possibility of using standard Prolog tools to analyze CLP programs arisen during discussions with John Gallagher and Andre de Waal from Bristol University in the context of the PRINCE project. Some experiments have been carried out in collaboration with them and other members of this project in order to evaluate the validity of this approach. Although the results of these experiments are to be discussed in more detail elsewhere we will summarize some of those obtained so far. In collaboration with Bristol a PrologIII lists interpreter has been developed which handles list equalities. Later, and in collaboration with PrologIA, a linear equation and disequation solver based on the Gaussian

algorithm was added, in order to handle also length constraints on lists and linear constraints. The meta-interpreter was then analized by some of the implemented analyzers we have currently available, namely: an abstract interpreter based on the "sharing" domain defined in [28] and another analyzer based on the "sharing + freeness" abstract domain defined in [27]. The results so far show that it is indeed sometimes possible to detect when a constraint program is being used in a "mode" which makes it (or part of it) a legal Prolog program and actually simplify the metainterpreted constraint solver accordingly. The resulting program when run under Prolog can execute in considerable less time than when running under some CLP systems. Of course, a CLP system which contained "Prolog builtins" and Prolog-type implementation technology for operations known to be on the Herbrand domain could achieve the same speed with this type of analysis. In quite complex cases, as with the list constraint solver, we have been less successful at inferring information about the original PrologIII program. This is mainly because the domains used, designed to be effective on Prolog programs which don't do extensive meta-interpretation are not detailed enough. We are planning on experimenting with finer domains and see if this brings better results. On the other hand, even in the cases of complex constraints, the analysis has been quite successful in simplifying and parallelizing the constraint solver, resulting in better performance in the metainterpretation.

3.2 Direct Analysis of Constraint Logic Programs

The obvious alternative to the analysis methods presented in the previous section is to analyze the constraint program itself, directly abstracting the semantics of the constraint operations. As far as we know, two frameworks for the analysis of constraint logic programs have been defined, by Marriott [25] and Codognet [4].

The former presents a framework in which top-down abstract interpretation can be developed. It is based in a definition-independent meta-language which can express the semantics of a wide variety of programming languages including constraint logic programming languages. The meta-language is one of typed lambda expressions with two types namely: static and dynamic. The static types interpreted as posets remains the same throughout all semantics, whereas dynamic types, interpreted as complete lattices, may change. Different interpretations provides different semantics from the same set of semantic equations. The dataflow semantics provided by non-standard interpretation is defined in such a way that developing a dataflow analysis consists in choosing a description domain *Acon* which captures the information required for the analysis, and defining appropriate base functions to approximate concrete ones.

The latter presents a general framework for the description of both constraint logic programming languages and their static analyses. First a general computation scheme, called *computation system* is defined by a 4-tuple $S = (C, \leq, \oplus, \pi)$, the computation domain, a partial order relation on C, a composition function from $C \times C$ to C and a projection function from $C \times 2^{\mathcal{X}}$ to restrict some element of C to a fixed set of variables $\mathcal{X}$, respectively. In this scheme S-programs and S-computations are defined. After that, the notion of simulation or abstraction between two computation systems $S = (C, \leq, \oplus, \pi)$ and $S' = (C', \leq', \oplus', \pi')$ is defined by the existence of a monotonic function γ from C' to 2^C. Therefore abstract S'-programs which perform the abstract interpretation of the S-programs can be derived. In addition, it allows multiple layers of abstraction, where a concrete system can be abstracted by another one which can be itself abstracted by another one and so on.

Both approaches are significant in that they provide a formal framework for the abstract interpretation of constraint logic programs. However, whatever the framework were, from a practical point of view a most important problem remains unsolved: that is, choosing the appropriate abstract domain which captures useful information about the constraint logic program and defining the abstract functions which approximate the concrete ones while losing little information, even if the constraint solver is simply modelled as a "black box".

In fact, the abstraction framework in which the abstract domain we will define is based, is that of Bruynooghe [1] extended to handle constraints. It can be done in a quite straightforward way since this framework abstracts mainly the control part, i.e. SLD-resolution, of logic programs leaving the abstraction domain and functions undefined. But the control part of a constraint logic program is similar to that of SLD-resolution, just extending unification with the concept of constraint satisfaction, in which unification is just one more type of constraint. Therefore we only have to define abstract

operations which approximate constraint satisfaction with appropriate concretization and abstraction functions. A more formal description of this extension of Bruynooghe's framework to the analysis of CLP programs can be found in [10].

4 Inference of Groundness Information

As we have seen abstract interpretation consists in choosing a description domain which captures the information required for the analysis, and defining appropriate abstract functions to approximate concrete ones. The aim of this section is to provide a general abstraction of constraint logic programming languages which makes it possible to achieve accurate groundness information. Two groundness analyses have been recently presented by [25] and [4]. The former presents an abstract domain which approximates the set of variables which are known to be definitely constrained to a unique value, i.e definitely ground. However this domain cannot accurately propagate groundness since no information about relations among variables is abstracted, i.e. if two variables X and Y have been involved in the constraint $X = Y$, and later X becomes ground, there is not possible to infer groundness propagation to Y. The latter presents a computation system (recall the framework summarized in 3.2) which is capable of accurate inferring groundness information by executing a transformed version of the original program in which groundness is approximated by propositional formulas, in a constraint logic programming language over the boolean domain.

Our objective is to propose an abstraction that is as accurate as the latter and to do so in a way that first, is as independent as possible from the particular constraint system used and second, can be executed in standard Prolog without any program transformation. Thus, our abstraction will be based on a high-level description of grounding patterns which is then easy to obtain for each particular type of constraint in an actual system.

Let us first present the notation which will be used from this point on. Let $\Pi = \Pi_C \cup \Pi_P$ be a set of predicate symbols as before. Let $\Sigma = \Sigma_H \cup \Sigma_C$ be a set of function symbols being Σ_H the Herbrand function symbols and Σ_C the rest, such that $\Sigma_H \cap \Sigma_C = \emptyset$. Let $\Re$ be the structure defined on (Π, Σ), I an $\Re$-interpretation, $\mathcal{X}$ a set of variables and Φ a non empty subset of $(\Pi_c, \Sigma \cup \mathcal{X})$-atoms closed under conjunction. We will represent a constraint system as $S = (\Re, I, \mathcal{X}, \Phi)$, a constraint logic programming language defined over it as L and a program in this language whose variables will be called $Pvar \subset \mathcal{X}$ as P. Finally we will represent an admissible atomic constraint of P in Φ as π, and the predicate symbol of π as $rel(\pi)$, being $rel(\pi) \in \Pi_C$.

Once the notation has been presented, we will informally discuss how capture groundness information. Two points have to be taken into account when approximating groundness information from a constraint logic program. First, the grounding characteristics of each type of constraint in the constraint logic programming language considered. Second, a representation which achieves accurate groundness propagation.

Let us explain the former with an example. Consider the following constraints in PrologIII:

$$X = Y.Z,\ X :: N,\ N > M,\ W = f(A, B).$$

in which "." represents list concatenation and '::" is a list length predicate symbol. In the first constraint grounding all variables but one, whatever they were, grounds the other. In the second constraint while grounding X grounds N, grounding N does not ground X. In the third constraint grounding one variable, whatever they were, does not ground the other. The fourth constraint is a special case of the first one. While it satisfies that grounding all variables but one grounds the other, it imposes an additional characteristic: grounding W grounds the rest of variables. In order to distinguish these kind of constraints, i.e those atomic constraints $\pi \equiv X = Y$ s.t. $X \in Pvar$, $Y \in (\Sigma_H \cup Pvar) - term$, we will call them atomic substitutions. Note that Y can only be formed by Herbrand function symbols, i.e. constraints like $X = f(Y + 3, Z)$ are not allowed. This can be done without loss of generality by separating the constraints into two atomic constraints (i.e. $X = f(W, Z), W = Y + 3$).

Therefore we can distinguish at least three kinds of atomic constraints w.r.t. their grounding characteristics:

- those about which we can ensure the groundness of one variable, whatever it was, from the groundness of the rest (which will be called fully grounding constraints),

- those about which we can ensure the groundness of some particular variables from the groundness of some other particular variables (which will be called partially grounding constraints),
- and those about which we cannot ensure the groundness of any variable in the constraint from the groundness values of any other variable.

Note that the first kind of constraints includes all atomic constraints which have "=" as predicate symbol, being the atomic substitution a special case.

Let us discuss briefly the second point to be taken into account, i.e. different forms of abstracting those grounding constraints. There are many ways of abstracting relations among variables established by constraints. The more accurate they are, the more complex both representation and abstract functions will be. For example, consider the following constraint, $X = Y + Z$. The most general abstract domain for approximating the relation established by it is {X,Y,Z}, i.e. a set of variables representing just that those variables "are related". Alternatively, in a more accurate abstract domain, the same representation could be concretized as "are in the same constraint". Consider now that the constraint $X = A + B$ is going to be added to the system. The abstract function representing it for the first domain would be the union of the elements of the sets, i.e. {X,Y,Z,A,B}, and in the second domain would be the union of the sets, i.e. {{X,Y,Z},{X,A,B}}.

While, in the first domain it is not possible to infer accurate groundness propagation, the second domain is powerful enough to infer it, but in a quite complicated way. Let A and B become ground. Since the second equation has only one variable, X becomes ground. As in the second domain each set represents each equation in the system, it is possible by transitivity to obtain the same result. It is necessary to detect that the set has become a singleton (i.e. an equation with only one variable) and thus X is ground. Then X can be eliminated from all other sets, keeping track of sets which have become singletons, and so on, resulting in the abstract constraint {{Y,Z}}.

It would be better to represent the constraints as a set of sets of variables in such a way that if one of the variables becomes ground we achieve groundness propagation by simply eliminating every set in which the variable appears. This has proved to be an advantage of the domains and abstract unification algorithms proposed in [28, 18, 27]. Therefore a variable would be ground if it does not appear in any set of the abstract constraint. Consider again the equation X = Y + Z. We have the following grounding information: grounding two variables grounds the others, and grounding just one variables does not ground the others. It would be abstracted as: given two variables, eliminating all sets in which at least one of them appears, eliminates the other one, and given one variable, eliminating all sets in which it appears eliminates neither of the other two.

Therefore we can infer that, first, neither X, nor Y, nor Z can appear in a set alone, second each variable has to appear in each set with at least one of the other variables and third, there must be a set for each couple in which the other variable does not appear. Therefore the least set of sets satisfying this is {{X,Y},{X,Z},{Y,Z}}. As we have seen atomic substitutions are one type of those fully grounding constraints, but with an additional condition. Consider the following atomic substitution $X = f(Y, Z)$, and the equation X = Y + Z. The abstraction for the equation was defined as {{X,Y},{X,Z},{Y,Z}}. Since the additional condition imposes that grounding X grounds the rest of variables, the set $\{Y, Z\}$ should be eliminated.

Once abstraction for fully grounding constraints has been explained, let us consider the abstraction for partially grounding constraints. Consider the following atomic constraint $X :: N$. We have the following grounding information: the groundness of X cannot be inferred from the groundness of any other variable in π, and grounding X grounds N.

Therefore, X has to appear in a set without any other variable, i.e. a singleton, and every set in which N appears has to have X as an element. The least set of sets satisfying this, is $\{\{X\}, \{X, N\}\}$. Note that we are talking only about atomic constraints, and therefore the constraint $X.Y :: N + M$ will be the conjunction of the three following atomic constraints: $X.Y = Z, N + M = W, Z :: W$.

4.1 Abstract Domain and Framework

Let us now formally define the rules for abstracting the groundness information about variables in an atomic constraint. Let π be the atomic constraint. For each variable $X \in var(\pi)$ we will construct the set $SS_X \subseteq \wp(\wp(var(\pi)))$, such that $S_X \in SS_X$ iff it satisfies that:

- grounding all $Y \in S_X$, grounds X.
- $\nexists S'_X$, $S'_X \subset S_X$ such that S'_X satisfies the above condition.

Each SS_X has the grounding information for X. Then we define a recursive function *combine* which takes as arguments two sets of sets of variables, RR (initially $\{\{X\}\}$) and SS_X, computing all possible combinations R, among X and one variable for each set of SS_X, which have accurate grounding information (i.e. those for which no other combination R' is a subset of R). Formally:

$$combine(RR, SS) = \begin{cases} if\ SS = \emptyset \quad then & \{R | \forall R \in RR, \nexists R' \in RR, R' \subset R\} \\ else & S \in SS, SS' = SS - \{S\},\ RR' = combine\_one(RR, S),\ combine(RR', SS') \end{cases}$$

where the function *combine_one* is defined as:

$$combine\_one(RR, S) = \begin{cases} if\ S = \emptyset \quad then & RR \\ else & \{\{R \cup Y\} |\ \forall R \in RR,\ \forall Y \in S\} \end{cases}$$

each $combine(\{\{X\}\}, SS_X)$ giving in the form informally discussed above the abstract groundness information for each X. Then the abstraction of π would be the union of the abstract information of all variables in π.

An abstract constraint Δ of the abstract domain Ground is an element of $D_\alpha = \wp(\wp(Pvar))$, which gives for a clause the set of sets of program variables in that clause which are *possibly* nonground and are related by grounding constraints.

Let again π be an atomic constraint and $var(\pi)$ the set of variables in π. The abstraction of the constraint π is defined as:

Definition 4 (Abstraction of an atomic constraint)

$$\Delta(\pi) = \bigcup_{X \in var(\pi)} combine(\{\{X\}\}, SS_X)\ \square$$

Let us explain it with some examples. Consider the atomic substitution X = f(Y,Z):

- For X, $SS_X = \{\{Y, Z\}\}$, and $combine(\{\{X\}\}, \{\{Y, Z\}\}) = \{\{X, Y\}, \{X, Z\}\}$.
- For Y, $SS_Y = \{\{X\}\}$, and $combine(\{\{Y\}\}, \{\{X\}\}) = \{\{Y, X\}\}$.
- For Z, $SS_Z = \{\{X\}\}$, and $combine(\{\{Z\}\}, \{\{X\}\}) = \{\{Z, X\}\}$.

therefore $\delta = \{\{X, Y\}, \{X, Z\}, \{Y, X\}, \{Z, X\}\} \equiv \{\{X, Y\}, \{X, Z\}\}$, which is the abstraction informally obtained before for atomic substitutions. Consider the atomic constraint X :: N :

- For X, $SS_X = \{\{\}\}$, and $combine(\{\{X\}\}, \{\{\}\}) = \{\{X\}\}$.
- For N, $SS_N = \{\{X\}\}$, and $combine(\{\{N\}\}, \{\{X\}\}) = \{\{N, X\}\}$.

therefore $\delta = \{\{X\}, \{N, X\}\} \equiv \{\{X\}, \{X, N\}\}$, which is again the abstraction informally obtained before for list length constraints. Consider now a partially grounding constraint defined as:

- For X, $SS_X = \{\{\}\}$, and $combine(\{\{X\}\}, \{\{\}\}) = \{\{X\}\}$.
- For Y, $SS_Y = \{\{\}\}$, and $combine(\{\{Y\}\}, \{\{\}\}) = \{\{Y\}\}$.
- For Z, $SS_Z = \{\{Y\}\}$, and $combine(\{\{Z\}\}, \{\{Y\}\}) = \{\{Z, Y\}\}$.

- For W, $SS_W = \{\{X,Y\},\{X,Z\}\}$ (note that the set $\{X,Z\}$ must appear since Y grounds Z), and $combine(\{\{W\}\},\{\{X,Y\},\{X,Z\}\}) = combine(\{\{W,X\},\{W,Y\}\},\{\{X,Z\}\}) = combine(\{\{W,X\},\{W,X,Z\},\{W,Y,X\},\{W,Y,Z\}\},\{\{\}\}) = \{\{W,X\},\{W,Y,Z\}\}$.

therefore $\delta = \{\{X\},\{Y\},\{Z,Y\},\{W,X\},\{W,Y,Z\}\}$ which accurately approximates the grounding information of the constraint. It is clear that, for all partially grounding constraints the set SS_X for each X should be defined by the abstract interpreter designer, from the grounding characteristics of each partially grounding constraint.

Let $\Delta 1$ and $\Delta 2$ be two abstract constraints. The partial order among them is defined as:

Definition 5

$$\Delta 2 \sqsubseteq \Delta 1 \textit{ iff } \Delta 1 \supseteq \Delta 2 \ \square$$

The function *lub* computes the least upper bound of two abstract constraints $\Delta 1$ and $\Delta 2$ by taking the union of their elements.

Definition 6 (lub)

$$lub(\Delta 1, \Delta 2) = \Delta 1 \cup \Delta 2 \ \square$$

The set of all tuples is a complete lattice w.r.t. $\sqsubseteq$, with top element $\top = \{\wp(\wp(Pvar)\}$ is in the domain. $\top$ contains no information, therefore it approximates the whole set of admissible constraints. Let $\bot$ be a new symbol and let $\forall \Delta \in Ground$, $\bot \sqsubseteq \Delta$ and $\neg\exists \Delta \in Ground$ such that $\Delta \sqsubseteq \bot$. Thus $(Ground \bigcup \{\bot\}, \sqsubseteq)$ is a complete lattice.

The abstraction function for a set (disjunction) of constraints is defined as:

Definition 7 (Abstraction of a set of constraints)

$$\alpha(\Pi) = \cup_{\pi \in \Pi} \Delta(\pi) \ \square$$

Let us give an intuitively idea of the concretization of an abstract constraint. Let Δ be an abstract constraint in Ground, Δ abstracts all sets of constraints π satisfying that:

- all program variables not present in any set of Δ are constrained to a unique value.
- if two variables X, Y which are not constrained to a unique value, do not appear in any set of the abstraction together, there is neither a fully grounding constraint involving them in π which is not an atomic substitution $Z = Term$ being X,Y $\in$ var(Term), nor a partially grounding constraint such that $\exists S_X$, $Y \in S_X$ and vice versa.

Definition 8 (Concretization of an abstract constraint)

$$\gamma(\Delta) = \{\pi \mid \pi \textit{ is an atomic constraint}, \alpha(\pi) \sqsubseteq \Delta\} \ \square$$

4.2 Abstract Conjunction Function

The abstract conjunction function is the dual of the abstract unification in the classical abstract analysis. The concrete operation is the conjunction $\wedge$, which takes as arguments the current constraint and an atomic constraint, and gives either the new constraint if it is consistent, or fails. The corresponding abstract operation Λ is a function which appliest to a tuple (δ, δ'), being δ the current abstract constraint and δ' the abstraction of the atomic constraint π which is going to be added , and gives the new abstract substitution $\Lambda(\delta, \delta') = (\Delta_f, \Delta_r, \Delta_s)$ in our abstract domain $\{Ground \cup \bot\}$. In order to provide a clear intuition of the abstract conjunction function definition we will give an informal idea of this definition.

1. For each program variable X which is not in $var(\pi)$ and it is not ground in δ we will add the singleton {X} to δ' giving δ''.

2. For each program variable X which is ground in δ but not in δ'' we will eliminate from δ'' each set in which X appears, obtaining δ_1, i.e. we will propagate the groundness in δ to the abstraction of π .

3. Then, for each variable X which is ground in δ_1 and it is not in δ, (i.e. those which have become ground) we will eliminate from δ each set in which X appears, obtaining δ_2 (i.e. propagating groundness to the current abstract constraint). Repeat steps 2 and 3 until neither δ_n nor δ_{n+1} change.

4. Finally, Δwe will be formed as:

   - each set $S \in \delta_{n+1}$, such that $\forall X \in S,\ \{X\} \in \delta_n$, i.e. all sets of δ_{n+1} about which no groundness information is added by δ_n.
   - each set $S \in \delta_n$, such that $\forall X \in S,\ \{X\} \in \delta_{n+1}$, i.e. all sets of δ_n about which no groundness information is added by δ_{n+1}.
   - the union of each couple of sets $S_n \in \delta_n, S_{n+1} \in \delta_{n+1}$, such that none of them are singletons (note that singletons in both abstractions have been added in the previous step), and their conjunction is not empty.

For example, let δ be the abstract constraint $\delta = \{\{X,Y\},\{X,Z\},\{Y\},\{Z\}\}$ and π the atomic constraint $\pi \equiv Y = Z$ which is abstracted as $\delta' = \{\{Y,Z\}\}$. The abstract constraint Δ will be computed by:

- $\delta'' = \{\{X\},\{Y,Z\}\}$
- $rec\_prop(\delta,\delta'') = (\delta,\delta'')$
- $\Delta = \{\{\}\} \cup \{\{Y,Z\}\} \cup \{\{X,Y,Z\}\}$

Auxiliary Functions

In order to simplify the notation of the definition of the abstract conjunction function we will define some additional auxiliary functions. The function *ground* takes two abstract constraints δ and δ' returning the set of program variables in δ' which are ground in δ. Formally,

$$ground(\delta,\delta') = \{X \mid \exists S' \in \delta', X \in S', \nexists S \in \delta, X \in S\}$$

The function *rec_prop* takes two abstract constraints δ and δ' propagating recursively the groundness from one to the other until a fixpoint is achieved. Formally,

$$rec\_prop(\delta,\delta') = \begin{cases} if\ prop(\delta,\delta') = \delta' & then\ \ (\delta,\delta') \\ else & rec\_prop(prop(\delta,\delta'),\delta) \end{cases}$$

where the function *prop* is defined as:

$$prop(\delta,\delta') = \{S' |\ S' \in \delta', S' \cap ground(\delta,\delta') = \emptyset\}$$

We will also define a function *add_rel* which takes as arguments two abstract constraints δ_1 and δ_2 and returns the result of adding each relation abstracted in the second argument to those abstracted in the first argument. Formally:

$$\begin{aligned} add\_set(\delta_2,\delta_1) = {} & \{S | \forall S \in \delta_2,\ s.t.\ \forall X \in S, \{X\} \in \delta_1\} \cup \{S' | \forall S' \in \delta_1,\ s.t.\ \forall Y \in S, \{Y\} \in \delta_2\} \cup \\ & \{S2 \cup S1 | \forall S2 \in \delta_2, \forall S1 \in \delta_1,\ s.t.\ \forall X,Y \in Pvar,\ S1 \not\equiv \{X\}, S2 \not\equiv \{Y\}, \\ & \quad S1 \cap S2 \neq \emptyset\} \end{aligned}$$

Let us now define formally the abstract conjunction function Λ. Let again δ be the current abstract constraint, π the atomic constraint and δ' the abstraction of π.

Definition 9 (Abstract conjunction function: $\Lambda(\delta, \delta')$)

$$\begin{aligned}\Lambda(\delta,\delta') \;=\; & \delta'' = \delta' \cup \{\{X\} | \forall X \in Pvar, X \notin var(\pi),\\ & (\delta_{n+1}, \delta_n) = rec\_prop(\delta, \delta''),\\ & add\_set(\delta_{n+1}, \delta_n)\end{aligned}$$

□

Let us illustrate the definition with an example. Consider the following part of a constraint logic program:

$....., p(X,Y,Z,W), Z = W, ...$

$p(X,Y,Z,W) : -Y \leq Z, X = Y + Z.$
$p(X,Y,Z,W) : -Y > Z, X = Y + W.$

Let $\delta = \{\{X\},\{Y\},\{Z\},\{W\}\}$ be the current abstract constraint before the execution of p/4, i.e all X, Y,Z and W are possibly non ground variables such that grounding any of them does not affect to the groundness of the rest of variables. In the first clause:

- the abstract conjunction of δ with the abstraction of $Y \leq Z$ will be result in $\delta 1 = \{\{X\},\{Y\},\{Z\}\{W\}\}$.
- the abstract conjunction of $\delta 1$ and the abstraction of $X = Y + Z$ results in $\delta 1' = \{\{X,Y\},\{X,Z\},\{Y,Z\},\{W\}\}$

On the other hand, in the second clause:

- The abstract conjunction of δ with the abstraction of $Y > Z$ results in $\delta 2 = \{\{X\},\{Y\},\{Z\},\{W\}\}$
- the abstract conjunction of $\delta 2$ and the abstraction of $X = Y + W$ results in $\delta 2' = \{\{X,Y\},\{X,W\},\{Y,W\},\{Z\}\}$

Finally, we compute the lub of $\delta 1'$ and $\delta 2''$ giving $Delta = \{\{X,Y\},\{X,Z\},\{X,W\},\{Y,W\},\{Y,Z\},\{Z\},\{W\}\}$. Then the abstract conjunction of Δ and the abstraction of the constraint $Z = W$ is computed by:

- $\delta'' = \{\{X\},\{Y\},\{Z,W\}\}$
- $rec\_prop(\delta, \delta'') = (\delta, \delta'')$
- $\Delta = \{\{X,Y\}\} \cup \{\{Z,W\}\} \cup \{\{X,Z,W\},\{Y,Z,W\}\}$

Recall that now grounding Z grounds W and vice versa, lefting X and Y depending on each other's groundness value.

5 Conclusions and Future Work

We have discussed some of the issues which arise in the dataflow analysis of constraint logic programming (CLP) languages using the technique of abstract interpretation. We have shown how a number of optimizations are possible in different CLP systems (including efficient parallelization) are presented and the information that has to be obtained at compile-time in order to be able to implement such optimizations discussed. Of the two approaches proposed for obtaining this information for a CLP program the one based on an analysis of a CLP metainterpreter using standard Prolog analysis tools was shown to be useful in some particular cases. However, a more detailed analysis of this approach

is necessary. The results of this analysis will be reported on elsewhere. For the second approach, the direct analysis of the CLP program, an abstract domain which approximates groundness (also referred to as "definiteness") information (i.e. constraint to a single value) and the related abstraction functions were presented. We are designing new abstract domains and abstract constraint conjunction algorithms for inferring other types of information such as freeness, type information, and entailment, and combinations of the previous domains.

References

[1] M. Bruynooghe. A Framework for the Abstract Interpretation of Logic Programs. Technical Report CW62, Department of Computer Science, Katholieke Universiteit Leuven, October 1987.

[2] M. Bruynooghe and G. Janssens. An Instance of Abstract Interpretation Integrating Type and Mode Inference. In *5th Int. Conf. and Symp. on Logic Prog.*, pages 669–683. MIT Press, August 1988.

[3] J.-H. Chang and Alvin M. Despain. Semi-Intelligent Backtracking of Prolog Based on Static Data Dependency Analysis. In *International Symposium on Logic Programming*, pages 10–22. IEEE Computer Society, July 1985.

[4] P. Codognet and G. Filé. Computations, Abstractions and Constraints. Technical report, INRIA, 1990.

[5] A. Colmerauer. Opening the Prolog-III Universe. In *BYTE Magazine*, August 1987.

[6] J. S. Conery. *The And/Or Process Model for Parallel Interpretation of Logic Programs.* PhD thesis, The University of California At Irvine, 1983. Technical Report 204.

[7] P. Cousot. Abstract Interpretation of Logic Programs. In *1991 International Conference on Logic Programming.* MIT Press, June 1991. Invted Talk.

[8] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conf. Rec. 4th Acm Symp. on Prin. of Programming Languages*, pages 238–252, 1977.

[9] P. Cousot and R. Cousot. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. Technical report, LIX, Ecole Polytechnique, France, 1991.

[10] M.J. Garcia de la Banda and M. Hermenegildo. Analyzing Constraint Logic Programs. Technical report, Computer Science Dept, Universidad Politecnica de Madrid, Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, September 1991.

[11] S. K. Debray and D. S. Warren. Detection and Optimization of Functional Computations in Prolog. In *Third International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pages 490–505. Imperial College, Springer-Verlag, July 1986.

[12] J. Gallagher and M. Bruynooghe. The Derivation of an Algorithm for Program Specialization. In *1990 International Conference on Logic Programming*, pages 732–746. MIT Press, June 1990.

[13] J. Gallagher, M. Codish, and E. Shapiro. Specialisation of Prolog and FCP Programs Using Abstract Interpretation. *New Generation Computing*, 6:159–186, 1988.

[14] F. Giannotti and M. Hermenegildo. A Technique for Recursive Invariance Detection and Selective Program Specialization. In *Proc. 3rd. Int'l Symposium on Parallel Language Implementation and Logic Programming*. Springer-Verlag, 1991.

[15] P. Van Hentenryck. Parallel Constraint Satisfaction in Logic Programming. In *Sixth International Conference on Logic Programming*, pages 165–180, Lisbon, Portugal, June 1989. MIT Press.

[16] M. Hermenegildo and F. Rossi. On the Correctness and Efficiency of Independent And-Parallelism in Logic Programs. In *1989 North American Conference on Logic Programming*, pages 369–390. MIT Press, October 1989.

[17] M. Hermenegildo and F. Rossi. Non-Strict Independent And-Parallelism. In *1990 International Conference on Logic Programming*, pages 237–252. MIT Press, June 1990.

[18] D. Jacobs and A. Langen. Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In *1989 North American Conference on Logic Programming*. MIT Press, October 1989.

[19] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. Technical report, Dept. of Computer Science Monash University, June 1986.

[20] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Fourteenth Ann. ACM Symp. Principles of Programming Languages*, pages 111–119. ACM Press, 1987.

[21] J. Jaffar and S. Michaylov. Methodology and Implementation of a CLP System. In *Fourth International Conference on Logic Programming*, pages 196–219. University of Melbourne, MIT Press, 1987.

[22] H. Simonis M. Dincbas and P. Van Hentenryck. Solving Large Combinatorial Problems in Logic Programming. *Journal of Logic Programming*, 8(1 & 2):72–93, 1990.

[23] H. Mannila and E. Ukkonen. Flow Analaysis of Prolog Programs. In *4th IEEE Symposium on Logic Programming*. IEEE Computer Society, September 1987.

[24] A. Marien, G. Janssens, A. Mulkers, and M. Bruynooghe. The Impact of Abstract Interpretation: an Experiment in Code Generation. In *Sixth International Conference on Logic Programming*, pages 33–47. MIT Press, June 1989.

[25] K. Marriott and H. Søndergaard. Analysis of Constraint Logic Programs. In *1990 North American Conference on Logic Programming*, pages 531–547. MIT Press, 1990.

[26] C.S. Mellish. Abstract Interpretation of Prolog Programs. In *Third International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pages 463–475. Imperial College, Springer-Verlag, July 1986.

[27] K. Muthkumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *1991 International Conference on Logic Programming*. MIT Press, June 1991.

[28] K. Muthukumar and M. Hermenegildo. Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. In *1989 North American Conference on Logic Programming*. MIT Press, October 1989.

[29] W. Older and A. Vellino. Extending Prolog with Constraint Arithmetic in Real Intervals. In *Canadian Conference on Electrical and Computer Engineering*, September 1990.

[30] H. Sondergaard. *Semantics-based Analysis and Transformation of Logic Programs*. PhD thesis, DIKU, Univ. of Copenhagen, 1990.

[31] A. Taylor. LIPS on a MIPS: Results from a prolog compiler for a RISC. Technical report, Association for Logic Programming, June 1990.

[32] P. Van Roy and A. M. Despain. The Benefits of Global Dataflow Analysis for an Optimizing Prolog Compiler. In *Proceedings of the North American Conference on Logic Programming*, pages 501–515. MIT Press, October 1990.

[33] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, Artificial Intelligence Center, SRI International, 333 Ravenswood Ave, Menlo Park CA 94025, 1983.

[34] R. Warren, M. Hermenegildo, and S. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Fifth International Conference and Symposium on Logic Programming.* MIT Press, August 1988.

[35] W. Winsborough. Path-dependent reachability analysis for multiple specialization. In *1989 North American Conference on Logic Programming.* MIT Press, October 1989.